VŠB - Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra Informatiky

Nezáporná dekonvoluce v reálném čase

Realtime nonnegative deconvolution

2009/2010 Michal Kravčenko

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Kravčenko**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Nezáporná dekonvoluce v reálném čase
Realtime Nonnegative Deconvolution

Zásady pro vypracování:

Některé obrazy bývají poškozeny konvoluční chybou, což může být způsobeno vlastností objektivu, špatným ostřením nebo dlouhým časem snímání. K odstranění těchto vad lze použít dekonvoluci. Její výpočet je rychlý, ale není odolná vůdči šumu. Proto se využívá podmínky nezápornosti výsledků, což ovšem vede k iterativnímu numerickému algoritmu. Pro využití nezáporné dekonvoluce v reálném čase je nutné vypočítat jednu iteraci během jedné milisekundy.

Obsahem práce bude zvolení vhodného algoritmu řešícího nezápornou dekonvoluci a jeho implementace do CUDA. S danou implementací bude provedeno měření závislosti výpočetního výkonu CUDA GPU na velikosti zpracovávaného obrazu, tak aby bylo zachováno zpracování v reálném čase.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Mgr. Štěpán Šrubař**

Datum zadání: 20.11.2009

Datum odevzdání: 07.05.2010

doc. Dr.Ing. Eduard Sojka
vedoucí katedry

prof. Ing. Ivo Vondrák, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně, a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

Chtěl bych tímto poděkovat svému vedoucímu Mgr. Štěpánu Šrubařovi za to, že mi umožnil se seznámit s dále popisovanými postupy a technologiemi a bez jehož rad a připomínek by tento text nebyl v prezentovatelné formě. Dále bych chtěl poděkovat své mamince za to, že má se mnou stále trpělivost a umožňuje mi studovat.

Abstrakt

Tato práce popisuje teorii konvoluce, nezáporné dekonvoluce a jejich algoritmické zpracování jednak sekvenčně, tak i paralelně. Pro paralelní výpočet byla specificky použita grafická karta podporující CUDA. Dále jsou zde představeny aplikace těchto kódů na reálné obrazy a byly změřeny veličiny jako relativní zrychlení a kvalita výpočtu.

Klíčová slova:

Konvoluce, dekonvoluce, nezáporná dekonvoluce, Goldova dekonvoluce, CUDA, paralelismus.

Abstract

This paper describes the theory of convolution, nonnegative deconvolution and their algorithmic realizations, both sequential and parallel. A graphic device supporting CUDA was used as a specific mean to realize the parallel computation. Furthermore, there are shown applications of those alghorithms on real life images and relative speed-ups and quality of solutions were measured.

Keywords:

Convolution, deconvolution, nonnegative deconvolution, Gold deconvolution, CUDA, parallelism.

Seznam použitých zkratek a symbolů

Zkratky

CUDA	-	**C**ompute **U**nified **D**evice **A**rchitecture
CPU	-	**C**entral **P**rocessing **U**nit
GPU	-	**G**raphic **P**rocessing **U**nit
DFT	-	**D**iscrete **F**ourier **T**ransform
FFT	-	**F**ast **F**ourier **T**ransform (Discrete)
iFFT	-	inverze FFT
MPRGP	-	**M**odified **P**roportioning with **R**educed **G**radient **P**rojections

Symboly

$//$	-	operátor dekonvoluce
$*$	-	operátor konvoluce
$\|\|x - y\|\|$	-	absolutní odchylka vektoru y od vektoru x
FFT(x)·FFT(y)	-	bodové násobení produktů FFT(x) a FFT(y)
FFT(x)/FFT(y)	-	bodové dělení produktů FFT(x) a FFT(y)
$\|x\|$	-	velikost x, počet prvků x
A^T	-	transponovaná matice A

Obsah

1 Shrnutí

Předmětem této práce je obeznámení s problematikou nezáporné dekonvoluce, s jejími různými formami a možnostmi realizace paralelních algoritmů na grafických adaptérech firmy NVIDIA podporujících CUDA.

Tento text je rozdělen do čtyř základních částí, ve kterých se postupně nahlíží na teoretické základy dekonvoluce, na popis CUDA, na možnosti paralelizace výchozího algoritmu, popřípadě jiné postupy pro paralelizaci, porovnání sekvenčních a paralelních kódů z hlediska rychlosti řešení a nakonec praktická ukázka aplikace algoritmu na reálný obraz.

Ve druhé kapitole tohoto textu se seznámíte s teorií konvoluce a dekonvoluce, zejména se zaměřením na popis Goldovy dekonvoluce, jež byla hlavním předmětem této práce. Dále je zde popsána technologie CUDA, její výhody, nevýhody, pro co je vhodná a pro co se nehodí.

Ve třetí kapitole je popsán sekvenční algoritmus řešící Goldovu dekonvoluci a možnosti jeho paralelizace. Dále je zde zamyšlení nad dalšími způsoby jak na výpočet nahlížet tak, aby byl rychlejší než dosavadní zpracování.

Ve čtvrté kapitole jsou znázorněny rozdíly mezi rychlostmi výpočtů Goldovy dekonvoluce sekvenčně na CPU a paralelně na CUDA v závislostech na veličinách jako jsou počet iterací, rozměry obrazu a rozměry konvolučního jádra.

V páté kapitole je pak znázorněn praktický výsledek aplikace popsaných algoritmů na reálnou černobílou fotografii.

2 Teorie

2.1 Konvoluce

Dekonvoluce je inverzním výpočtem konvoluce, a proto je pro její pochopení vhodné se nejdříve seznámit s konvolucí. Konvoluce je matematický operátor, podobně jako násobení, který ze dvou vstupních signálů vypočítá jeden signál výstupní. Tyto signály mohou být n-rozměrné. Jedna ze vstupních funkcí reprezentuje obraz a druhá tzv. konvoluční jádro, které daný obraz transformuje, přičemž nezáleží na tom, která z funkcí je chápána jako obraz a která jako jádro (konvoluce je komutativní, asociativní a distributivní). Obecná matematická rovnice pro jeden rozměr je následující:

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(\alpha) g(x - \alpha) d\alpha, \tag{1}$$

Diskrétní matematická rovnice pro jeden rozkměr je pak:

$$(f * g)[i] = \sum_{n=-\infty}^{+\infty} f[n] \cdot g[i - n], \tag{2}$$

kde f a g jsou vstupní funkce, $(f * g)$ je funkce výstupní. Jelikož my budeme uplatňovat tento vzorec při realizaci algoritmu pracujícího nad konečným obrazem, můžeme si předešlý vzorec přepsat na:

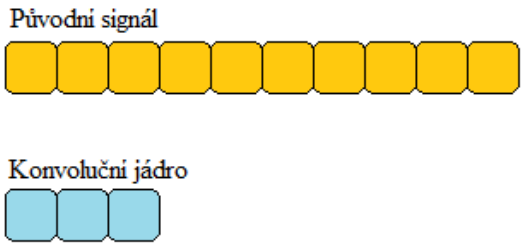
$$(f * g)[i] = \sum_{n=0}^{k-1} f[n] \cdot g[i - n], \tag{3}$$

kde velikost f je k, velikost g je l a nedefinované body g (v případě, že $i < n$) jsou rovny 0. Platí, že velikost $(f * g)$ je $k + l - 1$, což si pro usnadnění a použití v dalším textu označíme jako konvoluci vnější. Vnější konvoluce uchovává ve výstupu informace o ovlivnění vstupního signálu jak celým konvolučním jádrem, tak i pouhou částí jádra (1 až m prvků). Je třeba mít na paměti fakt, že takový výstupní signál je rozměrnější než signál vstupní. Dále stojí za zmínku konvoluce vnitřní, kdy na výstupu jsou pouze prvky vzniklé interakcí vstupního signálu s celým jádrem (m prvků) a jehož výsledný rozměr je $n - m + 1$.

Konvoluce se dá rozšířit na libovolný počet dimenzí. Vzorec pro dvourozměrný diskrétní signál je následující:

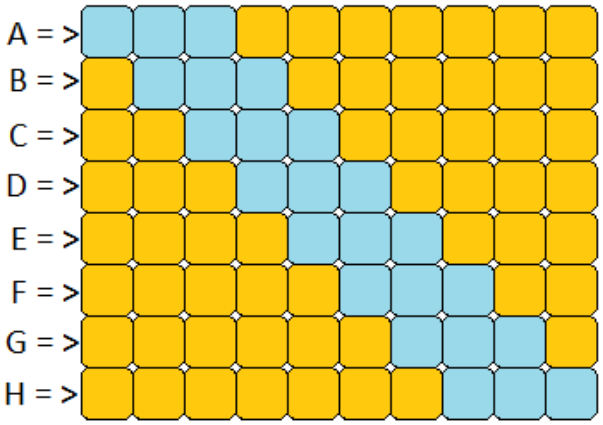
$$(f * g)[i, j] = \sum_{n=-\infty}^{+\infty} \sum_{m=-\infty}^{+\infty} f[n, m] \cdot g[i - n, j - m]. \tag{4}$$

Pro lepší představu o konvoluci zde předkládám názorné ilustrace reprezentující výpočet vnější a vnitřní konvoluce vstupního signálu o délce 10 pomocí konvolučního jádra o délce 3.



Obrázek 1. Reprezentace původního signálu a konvolucčního jádra.

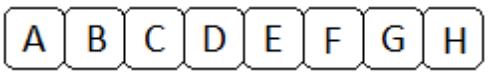
Průběh vnitřní konvoluce:



Obrázek 2. Průběh vnitřní konvoluce.

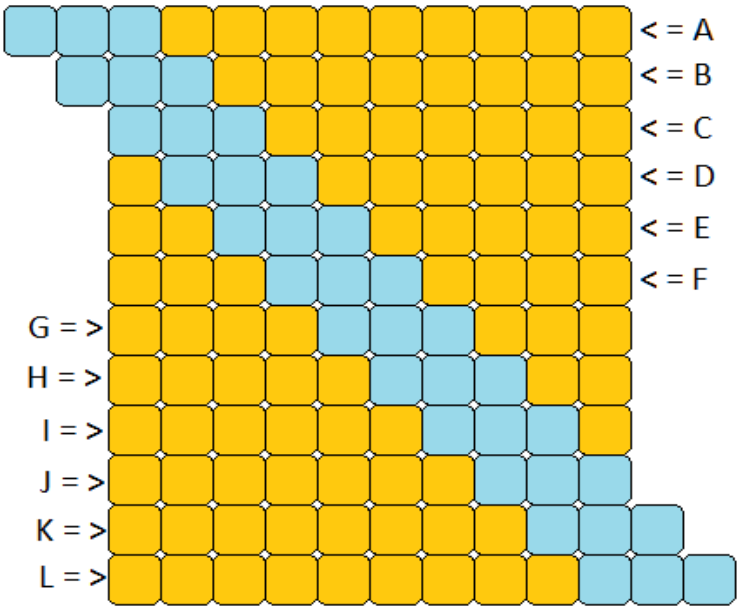
Výsledek vnitřní konvoluce je signál o rozměru 8 (10-3+1), kde prvek na pozici A je produktem řádku A v obrázku 2. Produktem řádku je myšleno:

- vynásobení hodnot odpovídajících si políček konvolučního jádra a původního signálu
- součet takto vypočítaných hodnot



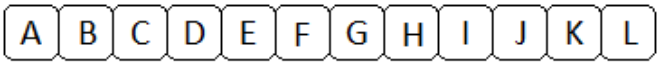
Obrázek 3. Výsledek vnitřní konvoluce.

Průběh vnější konvoluce lze znázornit následovně:



Obrázek 4. Průběh vnější konvoluce

Výsledek vnější konvoluce je v tomto případě signál o rozměru 12 (10+3-1), kde prvek na pozici A je produktem řádku A v obrázku 4. Produkt řádku je vypočítán obdobně jako v případě vnitřní konvoluce s tím rozdílem, že se neprovádí násobení mezi konvolučním jádrem a nedefinovanými hodnotami vstupního signálu.



Obrázek 5. Výsledek vnější konvoluce

Alternativně lze konvoluci spočítat pomocí diskrétní fourierovy transformace, což je asymptoticky rychlejší postup než v případě výše uvedeného vzorce 3. K tomu, aby se mezi sebou mohly vynásobit dva produkty fourierových transformací, je třeba, aby oba operandy měly stejnou velikost, jelikož je to násobení bodové. Toho lze docílit přidáním nul na okraje signálu s nižším počtem prvků. Fourierova transformace rozloží vstupní signál na množinu period nekonečných signálů, a předpokládá, že se tyto signály promítají do nekočna oběma směry (v případě jednorozměrného signálu), což je další důvod pro odsazení signálů nulami, čímž zamezíme projevu těchto periodicit. Jednorozměrná diskrétní fourierova transformace se vypočítá následovně:

$$X_k = \sum_{n=0}^{N-1} x_n \left(cos\left(\frac{2\pi}{N} nk\right) - i \cdot sin\left(\frac{2\pi}{N} nk\right)\right), \tag{5}$$

kde x je vstupní sekvence o N prvcích, X je produkt transformace o N prvcích, cosinus reprezentuje reálnou složku vstupního signálu a sinus reprezentuje složku imaginární.

Optimalizovaná verze výpočtu diskrétní fourierovy transformace se nazývá rychlá fourierova transformace, což je asymptoticky rychlejší postup, při kterém se počet operací sníží z $O(n^2)$ na $O(n \cdot \log n)$. Běžné implementace rychlé fourierovy transformace dosahují nejvyššího výkonu v případech, kdy je rozměr signálu dělitelný 2, čehož lze rovněž docílit přidáním nul na kraje signálů. Označme si rychlou fourierovu transformaci zkratkou FFT. Vzorec pro rychlý výpočet konvoluce je náslující:

$$(f * g) = iFFT(FFT(f) \cdot FFT(g)), \tag{6}$$

kde FFT(x) značí rychlou fourierovu transformaci signálu x a iFFT značí inverzi rychlé fourierovy transformace. Operátor · značí bodové násobení.

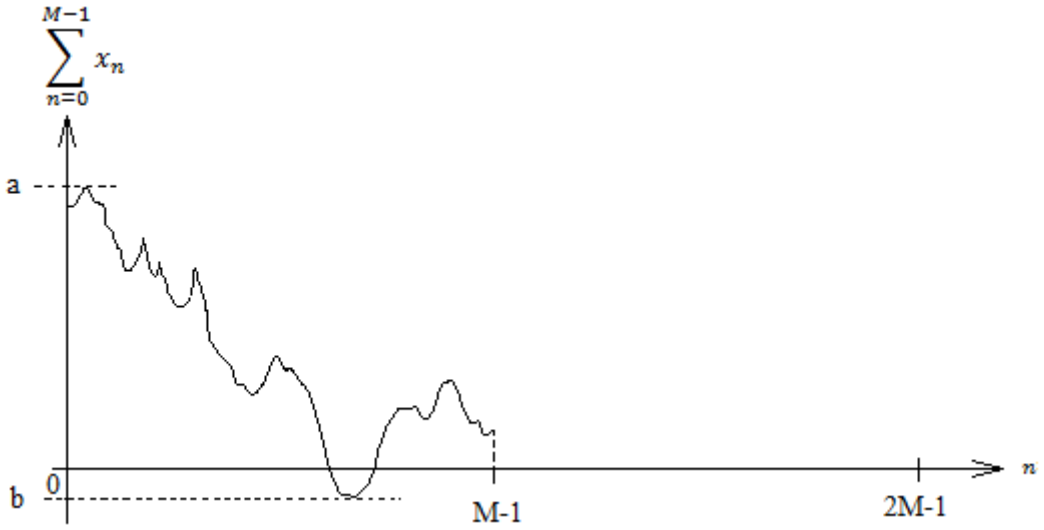
Konvoluce pomocí jednorozměrné fourierovy transformace pro dvourozměrné signály se počítá následovně.

- Spočítáme FFT pro každý řádek vstupu tak, jakoby každý řádek byl samostatný jednorozměrný signál.
- Výsledkem je jednorozměrná množina jednorozměrných signálů, čili jedno dvojrozměrné pole.
- Spočítáme FFT pro každý sloupec tohoto pole tak, že jednotlivé sloupce chápeme jako samostatné signály.
- Inverze se provádí obdobně, jen v opačném pořadí.

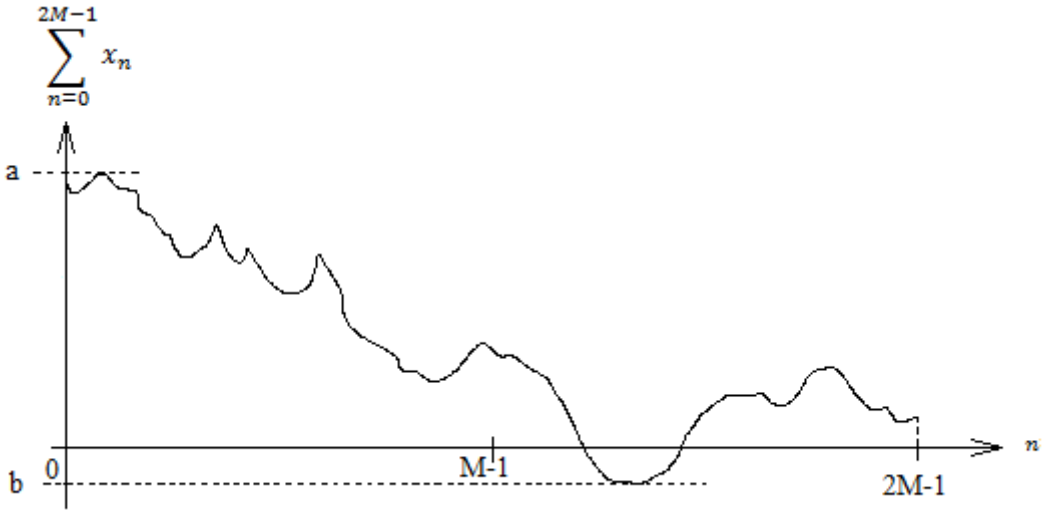
Pro dokázání, že můžeme signály odsazovat nulami před aplikací fourierovy transformace aniž by to ovlivnilo produkt konvoluce, můžeme vycházet z rovnice fourierovy transformace:

$$X_k = \sum_{n=0}^{N-1} x_n \left(cos\left(\frac{2\pi}{N} nk\right) - i \cdot sin\left(\frac{2\pi}{N} nk\right)\right). \quad (7)$$

N určuje počet vzorkovacích hodnot, k ovlivňuje fázový posun a x_n určuje amplitudu funkcí cosinus a sinus. Pokud bychom na kterýkoliv okraj x přidali hodnotu 0, pak by celková suma amplitud pro daný fázový posun nebyla ovlivněna, přičemž vzorkovací granuralita by se zvýšila o 1. Z čehož vyplývá, že obor hodnot funkce popisující transformovaný signál před a po přidání nuly by se nezměnil, jen definiční obor této funkce by se rozšířil. Tento princip si můžeme znázornit. Obrázek 6 popisuje neodsazený signál f o M prvích po aplikaci fourierovy transformace. Obrázek 7 popisuje výstup aplikace fourierovy transformace na signál f odsazeného M nulami (dvojnásobné zvýšení počtu prvků).



Obrázek 6. Znázornění výstupu fourierovy transformace na neodsazený signál f.



Obrázek 7. Znázornění výstupu fourierovy transformace na signál f odsazený M nulami.

Této vlastnosti fourierovy transformace lze tedy v případě konvoluce využít jak k rozšíření signálu s menším počtem prvků, tak k rozšíření obou signálů tak, aby byl výpočet diskrétní fourierovy transformace co nejrychlejší.

2.2 Dekonvoluce

Dekonvoluce je inverzní funknce konvoluce, pokud si rovnici konvoluce vyjádříme jako:

$$(f * g) + \varepsilon = h, \tag{8}$$

kde f je vstupní signál, g je konvoluční jádro, h je signál námi zachycený, ε je šum a operátor $*$ značí konvoluci. Pak naším cílem je nalézt co nejpřesnější hodnotu původního signálu. Je třeba vyřešit rovnici:

$$f = (h - \varepsilon)//g, \tag{9}$$

kde operátor $//$ značí dekonvoluci.

2.3 Dekonvoluce pomocí maticové násobení

Vzorec (1) popisující vnější konvoluci má po přepsání do maticového počtu tvar $y = K \cdot x$, který by po rozepsání mohl vypadat následovně:

$$\begin{bmatrix} y[0] \\ y[1] \\ \vdots \\ y[n+m-3] \\ y[n+m-2] \end{bmatrix} = \begin{bmatrix} k[0] & 0 & \cdots & 0 \\ k[1] & k[0] & \cdots & 0 \\ k[2] & k[1] & \cdots & 0 \\ \ldots & \ldots & \cdots & 0 \\ k[m-1] & k[m-2] & \cdots & 0 \\ 0 & k[m-1] & \cdots & 0 \\ \ldots & \ldots & \cdots & \ldots \\ 0 & 0 & \cdots & k[k0] \\ 0 & 0 & \cdots & \ldots \\ 0 & 0 & \cdots & k[m-2] \\ 0 & 0 & \cdots & k[m-1] \end{bmatrix} \cdot \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[n-2] \\ x[n-1] \end{bmatrix}, \tag{10}$$

kde x reprezentuje vstupní signál, K je konvoluční matice naplněná hodnotami z konvolučního jádra k tak, aby následné násobení odpovídalo definici (1) uvedené v kapitole popisující konvoluci, y reprezentuje výstupní signál.

Dekonvoluci tudíž můžeme spočítat jako:

$$\begin{bmatrix} k[0] & 0 & \cdots & 0 \\ k[1] & k[0] & \cdots & 0 \\ k[2] & k[1] & \cdots & 0 \\ \ldots & \ldots & \cdots & 0 \\ k[m-1] & k[m-2] & \cdots & 0 \\ 0 & k[m-1] & \cdots & 0 \\ \ldots & \ldots & \cdots & \ldots \\ 0 & 0 & \cdots & k[k0] \\ 0 & 0 & \cdots & \ldots \\ 0 & 0 & \cdots & k[m-2] \\ 0 & 0 & \cdots & k[m-1] \end{bmatrix}^{-1} \cdot \begin{bmatrix} y[0] \\ y[1] \\ \vdots \\ y[n+m-3] \\ y[n+m-2] \end{bmatrix} = \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[n-2] \\ x[n-1] \end{bmatrix}. \tag{11}$$

Matice K je čtvercová jen v případě, je-li délka konvolučního jádra rovna 1. Jelikož výpočty s takovým jádrem jsou triviální, není třeba jim věnovat zvláštní pozornost. Pro vysvětlení, v případě, kdy má konvoluční jádro jediný prvek, dosáhneme pouhého ztmavení ($k[0] < 1$), zesvětlení ($k[0] > 1$), či žádné změny obrazu ($k[0] = 1$). Ovšem na druhou stranu, pokud je délka jádra vyšší než 1, je třeba vypočítat inverzi, přesněji řečeno pseudoinverzi obdelníkové konvoluční matice, což je časově i paměťově velice náročný proces a v této práci jsem se touto cestou neubíral.

2.4 Dekonvoluce pomocí fourierovy transformace

Jelikož existuje metoda výpočtu konvoluce ve frekvenční doméně pomocí bodového násobení, pak je možno vypočítat i dekonvoluci ve frekvenční doméně pomocí bodového dělení. Produkt této operace je však signál obsahující komplexní hodnoty. Je to ovšem jednorázový výpočet a malá nepřesnost v konvolučním jádře či původních datech může způsobit velkou změnu ve výstupním signálu.

2.5 Nezáporná Dekonvoluce

Nezáporná dekonvoluce je postavená na konvoluční rovnici, jež je ekvivalentní s maticovým zápisem (10):

$$y = k * x, \tag{12}$$

kde y je obraz, na který chceme uplatnit dekonvoluci, k je konvoluční jádro a x je výsledný, upravený obraz, u kterého v průběhu celého výpočtu platí: $\forall \mathrm{p} \in \mathrm{x},\ \mathrm{p} \geq 0$ kde p je hodnota pixelu v rozmezí $\langle 0{,}255 \rangle$. Označme si operátor pro dekonvoluci jako $//$, potom rovnice pro dekonvoluci vypadá následovně:

$$x = y//k. \tag{13}$$

Zavedením kritéria nezápornosti budeme pracovat s reálnými hodnotami, které bychom při použití fourierovy transformace neobdrželi. Použití kritéria nezápornosti má ovšem za následek, že nejsme schopni nalézt přesné řešení dekonvoluční rovnice. Proto se nezáporná dekonvoluce definuje jako minimalizační úloha. Jejím úkolem je minimalizovat rozdíl mezi deformovaným signálem a konvolucí dosavadního řešení, čili minimalizovat

$$\big||y - k * x|\big|, \tag{14}$$

kde y je vstupní, deformovaný signál, k je konvoluční jádro a x je signál výstupní, který chceme co nejpřesněji aproximovat

V případě, že konvoluční jádro neznáme, popř. jej známe nepřesně, lze použít tzv. slepou dekonvoluci, kdy se v průběhu výpočtu aproximuje jak obraz, tak i jádro.

$$x = y//k,$$

$$k = y//x. \tag{15}$$

2.5.1 Algoritmy řešící nezápornou dekonvoluci

Za zmínku stojí 3 algoritmy řešící problém nezáporné dekonvoluce. Jejich stručné nastínění je uvedeno dále.

Metoda Lucy-Richardson

Dekonvoluce Lucy-Richardson je iterativní metoda používaná k řešení dekonvoluce a využívající přístupu iterativní minimalizace rozdílu. Její hlavní výhodou je její jednoduchost, jelikož předpokládá, že konvoluční signál známe zcela přesně. Pokud ovšem konvoluční signál známe jen částečně, je tato metoda nepoužitelná a je třeba se uchýlit ke složitějším metodám popsaných dále. Vzorec popisující Lucy-Richardson dekonvoluci je následující:

$$u_j^{(t+1)} = u_j^{(t)} \cdot \sum_i \frac{d_i}{c_i} p_{ij},$$

$$c_i = \sum_j u_j^{(t)} p_{ij}, \tag{16}$$

kde p je konvoluční signál, u reprezentuje signál, který chceme rekonstruovat, d je pozorovaný (ovlivněný) signál, c reprezentuje konvoluci signálu u a t značí index iterace.

Goldova dekonvoluce

Goldova dekonvoluce je rovněž iterativní metoda. Vychází ze základního konvolučního vztahu (9). Pro připomenutí:

$$\begin{bmatrix} \mathrm{y}[0] \\ \mathrm{y}[1] \\ \vdots \\ \mathrm{y}[\mathrm{n}+\mathrm{m}-3] \\ \mathrm{y}[\mathrm{n}+\mathrm{m}-2] \end{bmatrix} = \begin{bmatrix} \mathrm{k}[0] & 0 & \cdots & 0 \\ \mathrm{k}[1] & \mathrm{k}[0] & \cdots & 0 \\ \mathrm{k}[2] & \mathrm{k}[1] & \cdots & 0 \\ \ldots & \ldots & \cdots & 0 \\ \mathrm{k}[\mathrm{m}-1] & \mathrm{k}[\mathrm{m}-2] & \cdots & 0 \\ 0 & \mathrm{k}[\mathrm{m}-1] & \cdots & 0 \\ \ldots & \ldots & \cdots & \ldots \\ 0 & 0 & \cdots & \mathrm{k}[\mathrm{k}0] \\ 0 & 0 & \cdots & \ldots \\ 0 & 0 & \cdots & \mathrm{k}[\mathrm{m}-2] \\ 0 & 0 & \cdots & \mathrm{k}[\mathrm{m}-1] \end{bmatrix} \cdot \begin{bmatrix} \mathrm{x}[0] \\ \mathrm{x}[1] \\ \vdots \\ \mathrm{x}[\mathrm{n}-2] \\ \mathrm{x}[\mathrm{n}-1] \end{bmatrix}.$$

Pro vyřešní této soustavy lineárních rovnic využívá metodu nejmenších čtverců s omezením na nezáporné hodnoty. Úpravy vedoucí k obdržení vztahu pro výpočet dekonvoluce jsou následující:

$$y = K \cdot x\,. \tag{17}$$

$$K^T y = K^T K \cdot x\,.$$

$$1 = \frac{K^T y}{K^T K \cdot x}$$

$$x_i^{(j+1)} = \frac{(K^T y)_i}{(K^T K x^{(j)})_i} x_i^{(j)}, \tag{18}$$

kde x značí aproximaci signálu, který se snažíme zrekonstruovat, K je konvoluční matice vyplněná hodnotami z konvolučního jádra dle vztahu (9), y je vstupní poškozený signál a j značí index iterace. Goldova dekonvoluce byla zvolena jako algoritmus pro realizaci cílů této práce. Podrobnější popis a analýza jsou uvedeny dále v kapitole 3.

Metoda MPRGP

MPRGP je to algoritmus založený na metodě sdružených gradientů, která je využívaná k iterativnímu řešení soustav lineárních rovnic. V porovnání s metodou sdružených gradientů však zahrnuje tři alternativní kroky v každé iteraci. MPRGP uchovává v paměti dvě množiny výstupních hodnot, aktivní a pasivní. Pasivní množina obsahuje hodnoty, které nejsou limitovány hraničními podmínkami, zatímco v aktivní množině je zbytek. Algoritmus se pak v každé iteraci, v závislosti na hodnotě gradientu systému, rozhodne, jaký ze tří kroků použít. Tyto kroky jsou následující:

- sdružený gradient
- expanze
- změna poměru

Po vykonání zvoleného kroku jsou upraveny hodnoty v aktivní i pasivní množině výstupních hodnot a vypočítá se chyba systému lineárních rovnic. Výpočet lze ukončit v závislosti na hodnotě této chyby, proběhnutém výpočetním čase nebo počtu vykonaných iterací.

2.6 Architektura CUDA

2.6.1 Úvodní informace

CUDA je technologie pocházející z dílen laboratoří NVIDIA. Z porovnání rychlosti růstu výkonu desktop procesorů a grafických adaptéru vyplývá, že GPU mají vysoký výpočetní potenciál, donedávna využívaný převážně počítačovými hrami. NVIDIA vyvinula rozhraní umožňující programátorům využít tento výpočetní prostor.

CUDA si klade za cíl relativně snadné implementace paralelismů za použití již známých programovacích jazyků s minimálními přídavky v kódu, například za použití jazyka C. Z toho vyplývá, že se programátoři mohou soustředit na tvorbu samotného kódu a ne na hluboké studium nových vývojových prostředí či jazyků.

CUDA podporuje různorodé výpočty, kdy CPU se věnuje sekvenčním, či špatně paralelizovatelným a GPU se věnuje rozsáhlým, lehce paralelizovatelným výpočtům. To vede ke snadné implementaci již do funkčních kódů. V zásadě platí, že tam, kde chceme použít stejné posloupnosti instrukcí nad rozsáhlým souborem dat, tam je vhodné využít CUDA technologii.

Jednou z hlavních nevýhod této technologie je fakt, že doposud nedokáže pracovat s ukazateli, a díky tomu ani s dynamickými datovými strukturami. Z toho vyplývá, že ani rekurze není podporována, jelikož ta pro svou realizaci vyžaduje dynamický zásobník.

2.6.2 Obecný popis architektury

Zdrojový kód se skládá ze dvou částí. Klasického tzv. HOST kódu, což jsou bežné bloky instrukcí vykonávané CPU a tzv. DEVICE kódu, jež je spouštěn na GPU. Obě dvě části mohou být umístěny v jednom souboru a o jejich fyzické rozdělení se stará NVIDIA nástroj v průběhu kompilace.

DEVICE části kódu jsou v podstatě jen definice funkcí, procedur (v CUDA terminologii se nazývají kernely) a speciálních proměnných, které se definují stejně jako jejich CPU protějšky s tím rozdílem, že je třeba použít speciálních prefixů. Rozlišujeme mezi dvěma typy funkcí, kdy jeden je volatelný pouze z hostitelského kódu s návratovým typem void a druhý je funkce volatelná pouze z CUDA zařízení s libovolným návratovým typem. Při spouštění kernelu je třeba specifikovat kolik vláken, bloků vláken a v jakém geometrickém rozvržení se má úloze věnovat, přičemž každý kernel je schopen sám sebe identifikovat pomocí vestavěných proměnných a rozdělit tak přesně výpočetní práci. Volání kernelů z hostitelského kódu je výhradně asynchronní.

Práce s pamětí na GPU probíhá následovně. Nejdříve je třeba definovat ukazatele v hostitelském kódu s příslušným datovým typem a poté tento ukazatel použít ve speciální funkci alokující pamět na GPU. Nyní tento ukazatel slouží jako referenčním hodnota pro GPU nutná k rozeznání adresového prostoru, o který bychom eventuelně měli zájem při volání kopírovacích funkcí mezi CPU a GPU. Kopírování dat mezi CPU a GPU je možno v obou směrech a může být synchronní i asynchronní.

Obecný průběh programu běžícího na CUDA je pak následovný:

- Naplnění dat na CPU.
- Alokace dat na GPU.
- Kopírování dat z CPU na GPU.
- Asynchronní volání kernelu na GPU, který s daty něco provede.
- CPU nyní může počítat určitou podmnožinu daného problému, či cokoliv jiného, popřípadě jen čekat na výsledek.
- Kopírování dat z GPU na CPU.

CUDA je schopna spustit a udržet tisíce vláken pro naši potřebu. Pro efektivní využití všech jejich prostředků je třeba se podrobněji seznámit s jejím hardwarem a typickými vlastnostmi.

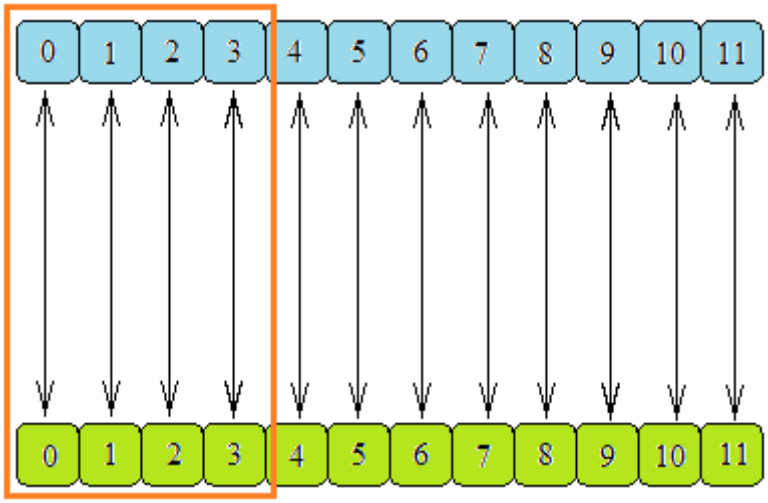
2.6.3 Architektura vláken

Každé CUDA zařízení má určitý počet multiprocesorových jednotek, z nichž každá je schopna obsluhovat osm warpů vláken v jeden okamžik. Warp je soubor 32 vláken vykonávající stejné instrukce v jednom instrukčním cyklu. Warpy jsou umístěny v instrukčních frontách, pro efektivní skrytí latencí při práci s pomalými pamětmi. V praxi vypadá činnost jednotky tak, že nejdříve inicializuje určitý počet vláken, sváže je do warpů a zatímco jeden warp načítá instrukce, druhý warp pracuje s daty a naopak.

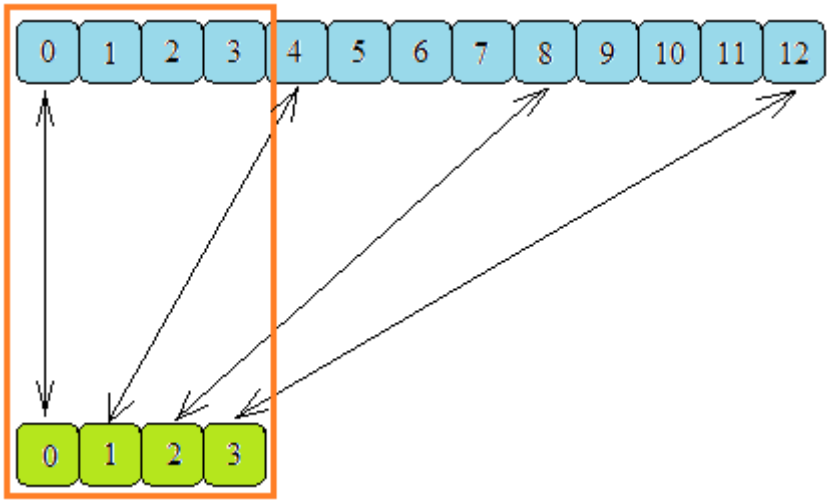
2.6.4 Paměti a jejich efektivní využití

Každé zařízení má k dispozici tři základní typy pamětí.

Globální paměť má vysokou kapacitu a obecně vysoké přístupové doby (až 600 cyklů). Používá se k ukládání dat z počítače a pro nahrávání dat z CUDA zařízení zpátky do počítače. Novější zařízení podporují techniku tzv. souběžného přístupu do paměti, což znamená, že v jednom cyklu jsou načteny i hodnoty sousedních adres (obr. 8). Proto je vhodné koncipovat algoritmy tak, aby k pamětem přistupovaly v tomto duchu a ne jinak (obr.9).



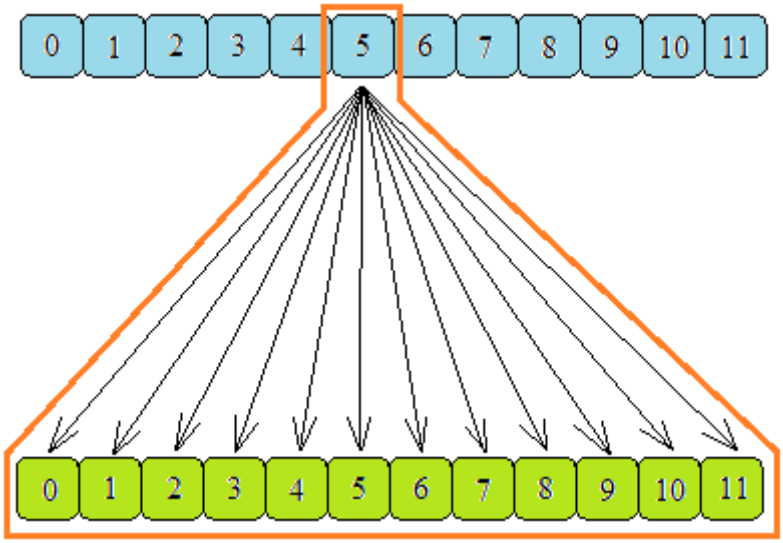
Obrázek 9. Grafické znázornění souběžného přístupu do paměti více vlákny v jeden paměťový cyklus. Modré buňky značí paměťovou buňku kde číslo uvnitř buňky je její adresa. Zelené buňky reprezentují vlákna a šipky mezi vlákny a pamětí znázorňují čtení/zápis dat. Oranžový obdelník znázorňuje jeden paměťový cyklus a která vlákna jsou v něm obsloužena.



Obrázek 10. Grafické znázornění jednoho typu nesouběžného přístupu do paměti více vlákny v jeden strojový cyklus. Je patrné, že je načtena jen jedna buňka a obslouženo jedno vlákno.

Část globální paměti, tzv. **lokální paměť**, se využívá k ukládání proměnných specifických pro každé vlákno v případě, že v rychlých registrech není dostatek místa. Proto je důležité abychom si byli vědomi paměťových nároků našich kernelů, aby k něčemu takovému nedošlo.

Konstantní paměť je rovněž část globální paměti, ovšem s optimalizací pro broadcast. Využívá malé cache, do které načte hodnotu z globální paměti a předá ji všem vláknům, které k ní v jednom cycklu přistupují. Je společná pro všechna vlákna a lze ji měnit jen z hostitelské části kódu. Je velice vhodné ji využít, pokud si jste jisti, že všechna vlákna budou v jeden okamžik načítat hodnotu ze stejné adresy, viz. obr. 10.



Obrázek 11. Efektivní využití konstatní paměti.

Sdílená paměť je společná pro všechna vlákna v jednom bloku, její kapacita je omezená a závislá na počtu multiprocesorových jednotek. Přístupová doba k ní je extrémně rychlá (pár cyklů) a je rovněž optimalizovaná pro souběžný přístup. Pokud ve svém kódu pracujete s hodnotou na jedné adrese více než jednou, tak je velice výhodné nejdříve všechna data, jež nás zajímají načíst do sdílené paměti, poté provádět početní operace a finální výsledek pak uložit do globální paměti. Je však nutné dát si pozor a práci správně rozvrhnout, jelikož ostatní bloky vláken nemají k této paměti přístup a při absenci synchronizací občas nastvává problém i uvnitř bloků samotných.

K efektivnímu využití sdílené paměti při řešení úloh pracujícími s velkými obory dat je vhodné popřemýšlet nad cestou jak řešení segmentovat tak, aby byl výsledek správný a sdílená paměť byla využitelná. Pokud jednoduchá segmentace stávajícího algoritmu není možná, je dobré se na problém podívat z jiného úhlu a zkusit najít sekvenční řešení, které bude sice pomalejší než originál, ale lépe paralelizovatelné.

Registry jsou prostory sloužící k ukládání parametrů kernelů a proměnných definovaných uvnitř kernelů. Jsou extrémně rychlé, podobně jako sdílená paměť a v případě překročení jejich kapacity je využita pomalá lokální paměť.

3 Nezáporná Goldova dekonvoluce

Nezáporná Goldova dekonvoluce byla zvolena jako prostředek pro realizaci praktické části této práce. V této kapitole budete seznámeni s realizací tohoto algoritmu jak sekvenčně, tak i paralelně. Pro připomenutí zde uvedu vztah (18) popisující tento výpočet:

$$x_i^{(j+1)} = \frac{(K^T y)_i}{(K^T K x^{(j)})_i} x_i^{(j)}.$$

Pro pochopení principu realizace je vhodné si výše uvedený vztah segmentovat a vysvětlit si co jednotlivé části znamenají:

- $K^T y = A$ značí aproximaci výstupního signálu (kterého chceme docílit) v závislosti na konvoluční matici K a vstupním signálu y.
- $K^T K x = D$ značí aproximaci výstupního signálu v závislosti na konvoluční matici K a aktuální hodnotě výstupního signálu x. Pro vysvětlení, $y = Kx$ (viz. vztah 17), tudíž $K^T K x = K^T y$. Také říká, že *x* v *j*-té iteraci je funkčně závislé na okolních hodnotách *x* v iteraci *j*.
- Poměr $A/D = C$ pak značí relativní odchylku aktuálního řešení D od referenční aproximace A.
- $C \cdot x$ pak tvoří návaznost na předchozí iteraci.

Pseudokód počítající Goldovu dekonvoluci by mohl vypadat například takto:

```
1.    výpočet B = K^T K
2.    pro každý řádek r vstupního obrazu
3.    {
4.          výpočet A = K^T (Y[r])
5.          naplnění X[r] počátečními hodnotami
6.          pro iteraci i nad daným řádkem r
7.          {
8.                výpočet D = B(X[r])
9.                pro každý prvek p řádku r výstupního obrazu
10.               {
11.                     výpočet X[r, p] = A[p]/D[p] · X[r, p]
12.               }
13.               kontrola ukončení výpočtu v závislosti na čase nebo
                  chybě systému lineárních rovnic
14.         }
15.   }
```

Kde $Y[r]$ značí vektor popisující r-tý řádek vstupu a $X[r]$ značí r-tý řádek výstupu. Pro vysvětlení řádku 5 musím uvést, že počátečními hodnotami je myšleno jakékoliv nezáporné číslo neblížící se příliš nule (pro omezení vlivu zaokrouhlování v prvních iteracích), číslo 1 je dobrá volba.

3.1 Sekvenční algoritmus

3.1.1 Nativní zpracování

V další části této kapitoly je popsáno řešení algoritmizace ednorozměrrné nezáporné vnější neslepé Goldovy dekonvoluce. Při realizaci řešení můžeme vycházet z nativní verze, které je pak možno dále optimalizovat. Definujme si rozměry vstupních veličin pro znázornění složitosti výpočtu této dekonvoluce. Je třeba uvést, že jednorozměrnou dekonvolucí dvojrozměrného obrazu je v tomto případě myšleno uplatnění jednorozměrné dekonvoluce nezávisle na každý řádek vstupního obrazu.

- p → délka konvolučního signálu
- m → délka řádku vstupního obrazu
- $n = m + p - 1$ → délka řádku výstupního obrazu
- r → počet řádků obrazu
- Matice K, reprezentující vliv konvolučního signálu na obraz, má počet řádků roven m a počet sloupců roven n.
- i → počet iterací

$A = K^T y$ provede nm operací násobení a $n(m-1)$ operací sčítání. Platí, že K se v průběhu celého výpočtu této dekonvoluce nemění, mění se pouze y a to v závislosti na řádku. Proto stačí A vypočítat jednou pro každý řádek.

$B = K^T K$ provede $n^2 m$ operací násobení a $n^2(m-1)$ operací sčítání. Platí, že K se v průběhu celého výpočtu nemění, a proto je nutné jej v průběhu výpočtu spočítat pouze jednou. B má n řádků a n sloupců.

$D = Bx$ provede n^2 operací násobení a $n(n-1)$ operací sčítání. Platí, že se tento výpočet provádí pro každý řádek a v každé iteraci.

$(A/D)\,x$ provede $2n$ operací násobení. Tento výpočet se rovněž provádí pro každý řádek a v každé iteraci.

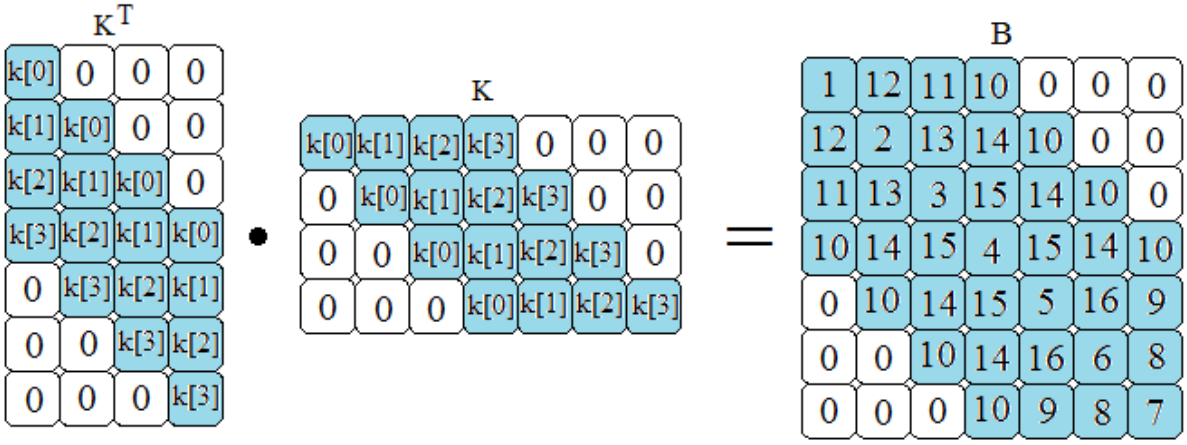
Celkový počet matematických operací je tedy roven $rn(2m - 1 + i(2n+1)) + n^2(2m-1)$. Asymptoticky je pak výpočetní složitost rovna $O(n^3)$, za předpokladu, že počet iterací se pohybuje v řádu desítek, m se neliší příliš od n a výška obrazu není daleko vzdálená od šířky obrazu, čili r je velice blízko m.

Paměťová náročnost nativní realizace tohoto algoritmu je pak rm pro y, rn pro x, mn pro K, n pro A, n^2 pro B a n pro D. Celkem tedy $r(m+n) + n(m+n+2)$ adresových polí v paměti.

3.1.2 Optimalizace nativního zpracování

Nativní úvaha o realizaci algoritmu je dobrý výchozí bod, nicméně by byla škoda nevyužít možnosti zamyslet se nad daným problémem trochu více a snížit tak paměťovou i časovou náročnost sekvenčního algoritmu.

Pro výpočet $B = K^T K$ není zdaleka nutné generovat matice K, K^T a poté je mezi sebou vynásobit. Platí, že tento výpočet vygeneruje omezený počet různých hodnot, přičemž my můžeme tento počet zrhuba určit, hodnoty vypočítat, uložit a později k nim přistupovat pomocí vhodného a nepříliš náročného indexování. Matice K je naplněna hodnotami z konvolučního jádra k, viz. (9). Na každém řádku K je posloupnost všech prvků konvolučního jádra k, kde počáteční prvek $k[0]$ je na hlavní diagonále a posloupnost se rozvíjí na pravo od něj V každém sloupci K^T je rovněž celá tato posloupnost, kde $k[0]$ je na hlavní diagonále a posloupnost se dále rozvíjí dolů. Grafické znázornění maticového vyjádření výpočtu B pro $|k| = 4, |y| = 4$ je znázorněno v obr. 11.



Obrázek 12. Maticové znázornění výpočtu B. k[x] udává hodnotu konvolučního jádra na pozici x. Hodnoty polí ve výsledné matici B udávájí index unikátního produktu násobení libovolného řádku z K^T s libovolným sloupcem z K (v tomto případě 16 unikátních hodnot, nulu nepočítaje).

Celkový počet prvků B je n^2, přičemž můžeme využít faktu, že prvky B jsou symetrické podle hlavní diagonály a v rozích mimo hlavní diagonálu jsou nulové hodnoty jejichž rozsah je závislý na délce vstupního řádku a na délce konvolučního jádra. Počet různých hodnot v B pak přibližne odpovídá $\sum_{i=0}^{p}(2i + 1) = \frac{p}{2}(p + 1)$, kde p je počet prvků v konvolučním jádře, což je o poznání méně než neoptimalizované B s redundantními informacemi o n^2 hodnotách. Při výpočtu $D = Bx$ pak navíc lze využít zmíněné symetrie a při polovičním počtu cyklů lze dospět ke správné hodnotě D.

Obdobný typ úvahy lze využít k optimalizaci výpočtu $A = K^T y$. Produkt tohoto výpočtu má n prvků a není nutné předem generovat matici K^T (lze pracovat jen s konvolučním jádrem o p prvcích). Pro každý tento prvek je nutné provést pouze p násobení a $p - 1$ sčítání. Výpočetní náročnost tedy klesne z $n(2m - 1)$ na $n(2p - 1)$.

3.2 Paralelizace sekvenčního algoritmu

Pro paralelizaci výše popsaného algoritmu je nutné si uvědomit, jak maximálně můžeme výpočet segmentovat tak, aby algoritmus produkoval shodné výsledky se svým sekvenčním protějškem. Dále je třeba vědět, kolik paměti je potřeba k výpočtu jednoho segmentu, což je informace nutná pro optimální sestavení dimenzí bloků vláken a vláken uvnitř každého bloku. Výhodný výchozí bod je sestavit paralelní kód produkující správný výsledek s nekomplikovaným přístupem do paměti, čímž mám na mysli přístup jen k paměti globální. Později je možno na tento relativně jednoduchý kód navázat a zamyslet se nad využitím pamětí rychlejších.

V případě výše popisované dekonvoluce (nezáporná, jednorozměrná) je nejmenší granularita, na kterou je možno výpočet rozdělit, granularita řádková. Je to proto, že řádky samy na sobě nejsou závislé, zatímco hodnoty výstupních pixelů jsou závislé na hodnotách sousedních řádkových pixelů v průběhu iterování. Proto jsem paralelní algoritmus koncipoval tak, aby se jednotlivé bloky vláken věnovaly samostatným řádkům (na jeden řádek připadal právě jeden blok). Další vlastností CUDA GPU je, že jeden multiprocesor dokáže spravovat maximálně 768 vláken, přičemž jeden blok má maximální kapacitu 512 vláken. Pro optimální využití kapacity všech multiprocesorů jsem tedy vlákna rozmístil tak, aby na každém multiprocesoru běžely dva bloky vláken kde každý blok obsahuje 384 vláken. Jelikož programově nejsme schopni ovlivnit na jakém multiprocesoru se daný blok vláken spustí, předpokládal jsem, že se CUDA sama postará o rozmístění bloků mezi multiprocesory.

Také je třeba zmínit, že zvolená technika paralelizace spočívala pouze v rozdělení vnějších cyklů mezi vlákna. Při výpočtu $B = K^T K$ se výpočtu účastnila všechna vlákna ve všech blocích. Výpočtu $A = K^T y$ a $D = Bx$, čili veličin vztažených k jednotlivým řádkům, se účastnil jeden blok vláken.

Dimenzi bloků vláken jsem nakonec nastavil na (2X, Y), kde X je počet multiprocesorů na GPU a Y je produkt funkce MIN(n, 384). Kde n je šířka výstupního obrazu a 384 je polovina maximálního počtu vláken udržitelných jedním multiprocesorem.

V další části tohoto textu je popsána realizovaná paralelizace sekvenčního kódu. Vstupní proměnné nutné pro výpočet a používané v uvedených kódech jsou:

- kernel - konvoluční jádro
- p - počet prvků konvolučního jádra
- vstup - vstupní obraz o rozměru $r \cdot m$
- vystup - výstupní obraz o rozměru $r \cdot n$
- m - počet pixelů na jednom řádku vstupního obrazu
- n - počet pixelů na jednom řádku výstupního obrazu $(m + p - 1)$
- r - počet řádků obrazu

Výpočet $B = K^T K$ a jeho paralelizace

Sekvenční kód:

```
1.    float sum;
2.    for (int i=0;i<p;i++){
3.          for (int j=0;j<p-i;j++){
4.                sum=0;
5.                for (int k=0;k<=j;k++){
6.                      sum+=kernel[k]*kernel[k+i];
7.                }
8.                B[i*(2*p-1)+j]=sum;
9.          }
10.         for (int j=p-i;j<2*p-1-2*i;j++){
11.               sum=0;
12.               for (int k=j-(p-i)+1;k<p-i;k++){
13.                     sum+=kernel[k]*kernel[k+i];
14                }
15.               B[i*(2*p-1)+j]=sum;
16.         }
17.   }

18.   int index = 0;
19.   for (int i=0;i<n;i++){
20.         Bd[i] = eval_d(B,n,p,i,i)
21.         for (int j=i+1;((j-i<p)&&(j<n));j++){
22.               Bod[index++] = eval_od(B,n,p,i,j)
23.         }
24.   }
```

Řádky 1-17 sekvenčního kódu představují výpočet unikátních hodnot vyskytujících se v *B*. Řádky 18-24 sekvenčního kódu představují reprezentaci hodnot *B* v takovém pořadí, v jakém budou načítány v iteračním cyklu počítající hodnoty výstupu a jsou zde zahrnuty pro omezení volání funkce *eval_d* a *eval_od* během těchto iterací. *Bd* uchovává data o prvcích na diagonále *B*, *Bod* pak o prvcích mimo diagonálu *B*. Fukce *eval_d* a *eval_od* slouží k reprezentaci matice *B* aniž bychom ji měli uloženou jako plnohodnotnou matici (využívá principů zmíněných v kapitole **3.1.2**).

Paralelní kód:

```
1.    int tid = threadIdx.x, threads = blockDim.x;
1.    for (int i = tid; i < n; i += threads){
2.          Bd[i] = eval_gpu_d(B,n,p,i,i)
3.          for (int j=i+1;((j-i<p)&&(j<n));j++){
4.                Bod[(j-i-1)*n+i] = eval_gpu_od(B,n,p,i,j)
5.          }
6.    }
```

Řádky 1-6 paralelního kódu odpovídají řádkům 18-24 sekvenčního kódu. Proměnná *threads* udává počet vláken v jednom bloku. Proměnná *tid* udává index specifického vlákna uvnitř bloku. Za zmínku stojí nutnost eliminace proměnné *index*, jelikož její správné uplatnění není možno zaručit (každé vlákno by začínalo s *index=0*) a nutnost jejího nahrazení dodatečným výpočtem. Funkce *eval_gpu_d* a *eval_gpu_od* jsou totožné se svými sekvenčními protějšky, jen s tím rozdílem, že jsou definovány pro spuštění z GPU.

Rozhodl jsem se sekvenční kód mezi řádky 1-17 neparalelizovat z toho důvodu, že bych byl schopný práci rozdělit pouze mezi *p* vláken (vnější cyklus má *p* iterací). Vzhledem k tomu, že *p*

obecně nabývá malých hodnot (několik desítek, v případě obrazů o rozumných velikostech), usoudil jsem, že bude výhodnější *B* spočítat na CPU a poté jej zkopírovat na GPU.

Kódy pomocných funkcí *eval_d* a *eval_gpu_d*:

```
1.    float eval_d(float *B, int n, int p, int x){
2.          if (x<p)
3.                return B[x];
4.          else if (x<n-p)
5.                return B[p-1];
6.          else
7.                return B[2*p-1-(n-x)];
8.    }
```

```
1.    __device__ float eval_gpu_d(float *B, int n, int p, int x){
2.          if (x<p)
3.                return B[x];
4.          else if (x<n-p)
5.                return B[p-1];
6.          else
7.                return B[2*p-1-(n-x)];
8.    }
```

Kódy pomocných funkcí *eval_od* a *eval_gpu_od*:

```
1.    float eval_od(float *B, int n, int p, int b, int y){
2.    int a = y - b;
3.    if (b<p-a)
4.          return B[a*(2*p-1)+b];
5.    else if (b<n-p)
6.          return B[a*(2*p-1)+p-1-a];
7.    else
8.          return B[a*(2*p-1)+2*(p-a)-1-(n-a-b)];
9.    }
```

```
1.    __device__ float eval_gpu_od(float *B, int n, int p, int b, int y){
2.    int a = y - b;
3.    if (b<p-a)
4.          return B[a*(2*p-1)+b];
5.    else if (b<n-p)
6.          return B[a*(2*p-1)+p-1-a];
7.    else
8.          return B[a*(2*p-1)+2*(p-a)-1-(n-a-b)];
9.    }
```

V případě výše uvedených pomocných funkcí používaných k reprezentaci virtuální matice *B* nebylo třeba nic paralelizovat z toho důvodu, že se tyto funkce skládají z pouhého větvení a nejsou v nich obsaženy žádné cykly.

Výpočet $A = K^T(Y[r])$ a jeho paralelizace

Sekvenční kód:

```
1.    for (int i=0;i<n;i++){
2.          sum=0;
3.          for (int j=p-1;j>=0;j--)
4.                if ((i-j>=0) && (i-j<m))
5.                      sum+=kernel[j]*vstup[i-j];
6.          A[i]=sum;
7.          vystup[i]=1;
8.    }
```

Řádky 1-8 sekvenčního kódu představují výpočet referenční řádkové konvoluce *A*. Na řádku 7 je výstupní řádek incializován na počáteční hodnotu.

Paralelní kód:

```
1.    __syncthreads();
2.    for(int i = tid;i<n;i+=threads){
3.          sum=0;
4.          for (int j=p-1;j>=0;j--){
5.                if ((i-j>=0) && (i-j<m)){
6.                      sum+=kernel[j]*vstup[i-j];
7.                }
8.          }
9.          vystup[i]=1;
10.         A[i]=sum;
11.   }
```

Řádky 1-11 paralelního kódu představují paralelní výpočet řádkové konvoluce *A*. Je nutné, aby se vlákna před samotným výpočtem synchronizovala, jelikož tento výpočet je prováděn uvnitř iterací upravujících samotný výstup. Bez synchronizace by se mohlo stát, že by se některá vlákna dostala k výpočtu výstupních hodnot aniž by zbývající vlákna dopočítala svůj díl *A*, jehož správná hodnota je nutná pro správnost výsledku. Na řádku 6 paralelního kódu je vidět, že vlákna přistupují k hodnotám pole *kernel* přes index *j*, jež není závislý na proměnné *i* iterující přes vnější cyklus. To znamená, že vlákna většinou přistupují ke stejné hodnotě pole *kernel* v jeden paměťový cyklus, což vybízí k využití konstantní paměti.

Pro jednoduché omezení práce s globální pamětí byla využita proměnná *sum*, do které byl načítán produkt násobení *kernel[j]*vstup[i-j]* namísto toho, aby byl přímo přičítán do globální proměnné *A*. Přístupů do globální paměti je tak vykonáno $n + 2np$ namísto $3np$.

Výpočet $D = B(X[r])$ a jeho paralelizace

Sekvenční kód:

```
1.    index = 0;
2.    for (int i=0;i<n;i++){
3.          D[i]=0;
4.    }
5.    for (int i=0;i<n;i++){
6.          D[i]+=Bd[i]*vystup[i];
7.          for (int j=i+1;((j-i<p)&&(j<n));j++){
8.                D[i]+=Bod[index]*vystup[j];
9.                D[j]+=Bod[index]*vystup[i];
10.               index++;
11.         }
12.   }
```

Řádky 2-4 představují inicializaci pole *D*. Řádky 5-12 představují naplnění *D* za použití virtuální matice *B*.

Paralelní kód:

```
1.    __syncthreads();
2.    for (int i= tid;i<n;i+=threads){
3.          sum = Bd[i]*vystup[i];
4.          for (int j=i+1;((j-i<p)&&(j<n));j++){
5.                sum +=Bod[(j-i-1)*n+i]*vystup[j];
6.          }
7.          D[i] = sum;
8.    }
9.    __syncthreads();
10.   for(int j= tid;j<n;j+=threads){
11.         iref = max(0,j-p+1);
12.         sum = 0;
13.         for(int i=iref;i<j;i++){
14.               sum +=eval_gpu_od(A,n,p,i,j)*vystup[i];
15.         }
16.         D[j] += sum;
17.   }
```

Paralelizace *D* musela být rozdělena do dvou kroků. Řádky 1-8 paralelního kódu představují výpočet hodnot *D[i]* odpovídající řádkům 6 a 8 v sekvenčním kódu. Řádky 9-17 představují výpočet hodnot *D[j]* odpovídající řádku 9 v sekvenčním kódu. Výpočet *D* v paralelním kódu bylo nutné rozdělit z toho důvodu, že při přístupu vlákna *i* k paměti na pozici *D[j]* docházelo ke konfliktu (s pamětí na pozici *D[j]* zároveň pracovalo vlákno *j*). Pro odstranění tohoto konfliktu bylo nutné řádek 9 sekvenčního kódu separovat a umístit jej do samostatného obslužného bloku. Pro správnost tohoto řešení bylo nutné obslužné cykly *for* předefinovat tak aby vnější cyklus iteroval přes proměnnou *j*. Tím bylo dosaženo toho, že hodnotou *D[j]* bylo manipulováno pouze vláknem *j*. Přidání dodatečných cyklů však způsobilo snížení rychlosti řešení. Přítomnost synchronizace na řádku 1 je nutná z toho důvodu, že výpočet *D* je závislý na hodnotě proměnné *vystup* v dané iteraci. Bez této synchronizace by se mohlo stát, že by některá vlákna počítala s neplatnými hodnotami proměnné *vystup*. Synchronizace na řádku 9 v paralelním kódu zamezuje výskytu konfliktů, kvůli kterým jsme provedli výše zmíněnou separaci. Dalším důvodem pro použití synchronizací je zaručení souběžného přístupu do paměti všemi vlákny.

Výpočet $\mathbf{X}[\mathbf{r},\mathbf{p}] = \mathbf{A}[\mathbf{p}]/\mathbf{D}[\mathbf{p}] \cdot \mathbf{X}[\mathbf{r},\mathbf{p}]$ a jeho paralelizace

Sekvenční kód:

```
1.    for (int i=0;i<n;i++){
2.          if (abs(D[i])<0.00001)
3.                vystup[i]=0;
4.          else
5.                vystup[i]*=A[i]/D[i];
6.    }
```

Řádky 1-6 sekvenčního kódu představují výpočet hodnot výstupu pro novou iteraci.

Paralelní kód:

```
1.    __syncthreads();
2.    for(int j= tid;j<n;j+=threads){
3.          if (abs(D[i])<0.00001)
4.                vystup[i]=0;
5.          else
6.                vystup[i]*=A[i]/D[i];
7.    }
```

Řádky 1-7 paralelního kódu představují paralelní výpočet hodnot výstupu pro novou iteraci. Z důvodu jednoduchosti této části kódu a tudíž absencí různých konfliktů nebyla paralelizace komplikovaná. Synchronizace na řádku 1 je nutná, jelikož pro správnost řešení je nutné aby nejdříve byly spočítány všechny hodnoty *D*.

3.3 Možnosti rozvoje a vylepšení

V době, kdy byla tato práce dokončována a připravována k tisku, byl algoritmus ve funkčním stádiu, ovšem ne ve stavu takovém, se kterým bych byl spokojený.

Vzhledem k povaze sekvenčního kódu realizujícího výpočet Goldovy dekonvoluce nebylo možné jednoduše segmentovat výpočet na nižší úroveň než výpočet jednoho řádku. Pro správný výpočet jednoho řádku je totiž nutné v každé iteraci postupně upravit všechny výstupní hodnoty. Všechny tyto hodnoty jsou na sobě navíc funčně závislé. Paměťová náročnost výpočtu takového řádku nedovolovala využití sdílené paměti. Jedno řešení bych viděl v možnosti rozdělení řádku na díly tak, aby se výpočet jednoho takového dílu vešel do sdílené paměti jednoho mikroprocesoru. Takový přístup by ale měl, díky absenci návaznosti na správné hodnoty sousedních pixelů v sousedních segmentech, za následek horší kvalitu výstupu. Nicméně je tento přístup možná varianta, jelikož využití sdílené paměti by dle mého názoru poskytlo nemalé urychlení. Skutečné zhoršení kvality stojí za prozkoumání, popřípadě je možno popřemýšlet o jiném postupu ubírajícího se tímto směrem.

Dalším nedostatkem je to, že výše popsaný algoritmus je jednorozměrný a ve skutečném světě se jednorozměrná obrazová dekonvoluce téměř nevyskytuje. Je proto nutné přepracování výchozího vztahu do dvourozměrné podoby. Jedno řešení takového problému může být aplikace jednorozměrné dekonvoluce na řádky a poté na sloupce (či naopak). Takový triviální přístup je však použitelný jen pokud je konvoluční jádro symetrické podle obou diagonál. Skutečné dojrozměrné řešení by mělo za následek, že by se funční závislost pixelů na sousedních pixelech rozšířila i na sloupce, což by způsobilo, že pro zcela správný výpočet by se v jeden okamžik musela v paměti uchovat data popisující tolik řádků vstupu a výstupu, kolik řádků by mělo konvoluční jádro. To by nebyl sám o sobě problém, jelikož globální paměti má CUDA k dispozici dostatek, ale zcela určitě by to znemožnilo využití sdílené paměti.

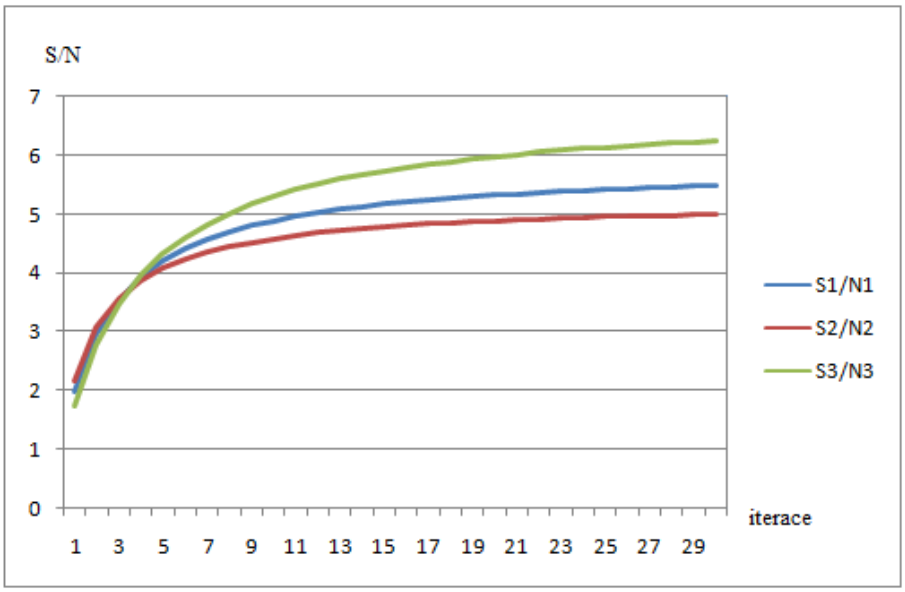
Dalším nedostatkem je absence odstranění artefaktů způsobených samotnou dekonvolucí. Aplikace dekonvoluce na obraz má za následek nejen zaostření ale i dodatečný efekt, který bych mohl popsat jako rozezvonění hran v obrazu, o čemž se ostatně můžete přesvedčit z obrázků 15 a 16 v kapitole 4. Interval mezi těmito „ozvěnami“ je přímo úměrný délce konvolučního jádra. Díky periodickému charakteru těchto artefaktů se nabízí využití fourierovy transformace k jejich identifikaci a redukci, popřípadě eliminaci.

Dalším možným rozšířením je realizovat řešení slepé dekonvoluce. Slepá dekonvoluce dokáže opravit obraz, aniž by znala přesné hodnoty konvolučního jádra, přičemž pokud známe jádro aslespoň částečně, tak průběh řešení konverguje rychleji. Slepá dekonvoluce spočívá v upravování matice K v iteračních cyklech společně s výstupem x. Vztah pro výpočet slepé dekonvoluce by pak vypadal následovně:

$$1.\quad X[r,i] = \frac{\left(K^T(Y[r])\right)[i]}{\left(K^T K(X[r])\right)[i]} X[r,i],$$
$$2.\quad K^T[i,j] = \frac{(xy^T)[i,j]}{(xx^T K^T)[i,j]} K^T[i,j]. \tag{18a}$$

Kroky 1 a 2 se vykonají jeden po druhém uvnitř iterací algoritmu, přičemž na pořadí nezáleží.

Toto navrhované řešení slepé dekonvoluce je však extrémně výpočetně i paměťově náročné. Zatímco během „neslepé“ dekonvoluce se mění pouze hodnota x, během slepé dekonvoluce se mění i K. To by v praxi znamenalo nutnost přepočítat **v každé iteraci** všechny parametry související s výpočtem, B, A i D. Celková výpočetní náročnost by pak vzrostla z $rn(2m-1+i(2n+1))+n^2(2m-1)$ na $in[n(2m-1)+2r(m+n)+(2r-1)(2m+n)]$.



Obrázek 13. Porovnání náročnosti výpočtu jednorozměrné dekonvoluce neslepé s dekonvolucí slepou v závislosti na počtu iterací. S = slepá dekonvoluce, N = neslepá dekonvoluce. Vlastnosti obrazu v poměru 1: m=1000, p=50, r=1000. Vlastnosti obrazu v poměru 2: m=500, p=50, r=1000. Vlastnosti obrazu v poměru 3: m=1000, p=50, r=500.

Slepou dekonvoluci jsem realizoval ve fázi, kdy byl algoritmus ve stádiu nativní implementace maticového počtu a kdy byly v jedné iteraci prováděny výpočty všech řádků výstupu x a všech řádků matice K. K se v průběhu výpočtu blížila hodnotám, ze kterých vycházela konvoluce. Když jsem pak podobný postup uplatnil v optimalizovaném řešení, kdy se řádky počítaly nezávisle na sobě, pak slepá dekonvoluce neprobíhala správně. Z pozorování jsem usoudil, že hodnoty konvolučního jádra se aproximovaly na následujícím způsobem. Nechť je obsah konvolučního jádra aplikovaného na obraz $[n_0, n_1, \cdots, n_{x-1}, n_x]$, kde $(\sum_{i=0}^{x} n_i) = 1$, pak obsah konvolučního jádra aproximovaného během slepé dekonvoluce počítající řádky nezávisle na sobě byl $[a, a, \cdots, a, a,]$, čily všechny prvky aproximovaného jádra byly totožné. Z toho jsem usoudil, že s čím více řádky je počítáno v jedné iteraci, tím přesnější a rychleji konvergující slepou dekonvoluci obdržíme. Pro upřesnení by bylo vhodné přepsat část vztahu odpovídající aproximaci K takto:

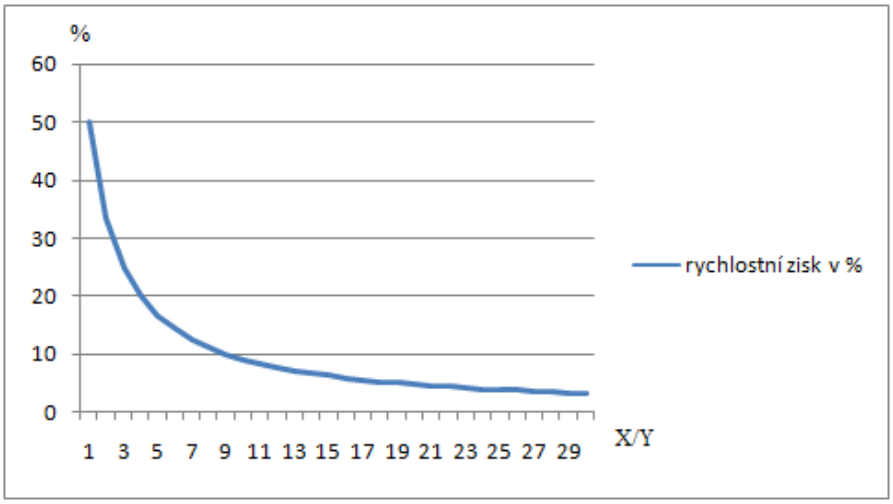
$$K^T[i,j] = \frac{(XY^T)[i,j]}{(XX^TK^T)[i,j]} K^T[i,j]. \qquad (18b)$$

Kde X a Y již nepředstavují samostatné řádky x a y, nýbrž soustavu řádků, nejlépe celý obraz. To ovšem představuje problém pro paralelní zpracování, jelikož dosavadní řádkově nezávislé zpracování je již dost paměťové náročné a počítání s více řádky najednou by urychlení paralelizace jen více ztížilo. Z důvodu výskytu výše popsaných netriviálních komplikací jsem slepou dekonvoluci do řešení nezačlenil.

Pseudokód popisující navrhovanou slepou dekonvoluci:

```
1.    naplnění X počátečními hodnotami
2.    pro iteraci i nad daným výstupem X{
3.          výpočet B = K^T K
4.          pro každý řádek r vstupního obrazu Y{
5.                výpočet A = K^T(Y[r])
6.                výpočet D = B(X[r])
7.                pro každý pixel j výstupního řádku r
8.                      výpočet X[r, j] = A[j]/D[j] · X[r, j]
9.          }
10.         výpočet E = XY^T
11.         výpočet F = XX^T
12.         výpočet G = FK^T
13.         pro každý řádek n matice K
14.               pro každý sloupec m matice K
15.                     výpočet K^T[m, n] = E[m, n]/G[m, n] · K^T[m, n]
16.   }
```

Poslední a jeden z nejsnadněji realizovatelných triků zvyšující rychlost výpočtu spočívá v rozdělení výpočtu mezi CPU a GPU. Následující postup se dá uplatnit pro obecný CUDA kód. V případě, že jsme s kódem spokojeni, je možno si spočítat rychlost výpočtu na GPU a rychlost výpočtu na CPU, poté pak výpočet rozdělit tak, aby CPU spočítalo část úlohy a skončilo v přibližně stejný čas jako výpočet podúlohy na GPU. Pro znázornění řekněme, že výpočet úlohy trvá X jednotek času na CPU nebo Y jednotek času na GPU. Pokud bychom výpočet prováděli pouze na GPU, pak by CPU čekal na výsledek Y jednotek času. Pokud se ovšem rozhodneme zaměstnat i CPU, pak maximální zrychlení výpočtu bude odpovídat $(1 - \frac{YX}{Y+X}) \cdot 100\%$, přičemž GPU bude počítat $\frac{X}{X+Y} \cdot 100\%$ a CPU $\frac{Y}{X+Y} \cdot 100\%$ dané úlohy. V reálném světě však tyto ideální stavy neplatí a tak bude zrychlení takto získané nižší v závislosti na ceně přesunu dat mezi CPU a GPU, na ceně správy vláken, předávání řízení CPU atd. Graf (obr. 13) znázorňující ideální zrychlení získané tímto způsobem je uveden níže.



Obrázek 14. Znázornění ideálního zrychlení získaného rozdělením výpočtu obecné úlohy mezi CPU a GPU v závislosti na poměru mezi X a Y. Z grafu vyplývá, že čím rychlejší je paralelní kód v porovnání s kódem sekvenčním, tím méně efektivní toto výpočetní rozložení bude. V mém případě, kdy je paralelní kód 2x-4x rychlejší by ideální zrychlení dosahovalo 20%-33%.

3.4 Problémy při realizaci paralelizace

Důležitou součástí samotného vývoje paralelního algoritmu je, dle mého názoru, schopnost určit správnost řešení. V ranných fázích vývoje jsem se bohužel vypravil cestou „celé to udělám, pak se uvidí“, přičemž jsem nepředpokládal, že samotný proces paralelizace bude tak komplikovaný jak se nakonec ukázal být. To mělo za následek dlouhé noci strávené hledáním chyb. V důsledku toho jsem se rozhodl postupovat znovu od začátku a tentokrát systematicky. Algoritmus jsem rozdělil na jednotlivé nezávislé segmenty. Pro každý segment jsem sestrojil odpovídající paralelní kód, načež testování samotné probíhalo tak, že jsem náhodně vygeneroval vstupní hodnoty nutné pro výpočet daného segmentu, poté jsem dané segmenty s těmito hodnotami spustil a zkoumal odchylky od sekvenčního kódu. Tento přístup se ukázal být přínosný a vývoj správného paralelního kódu byl tak relativně bezproblémový.

Jeden z problémů, jehož identifikace se ukázala být jedna z obtížnějších, byla nutnost psaní kódu v jazyce C a ne C++. Vytvořil jsem jednu z prvních verzi paralelního kódu, který produkoval špatné výsledky a přitom algoritmus samotný vypadal správně. Chyba se ukázala být v přístupu, jakým jsem iteroval jedním cyklem while. Pro názornost zde uvedu princip této chyby.

Špatná iterace cyklem while:

```
1.    while (promenna--){
2.          ...
3.    }
```

Správná iterace cyklem while:

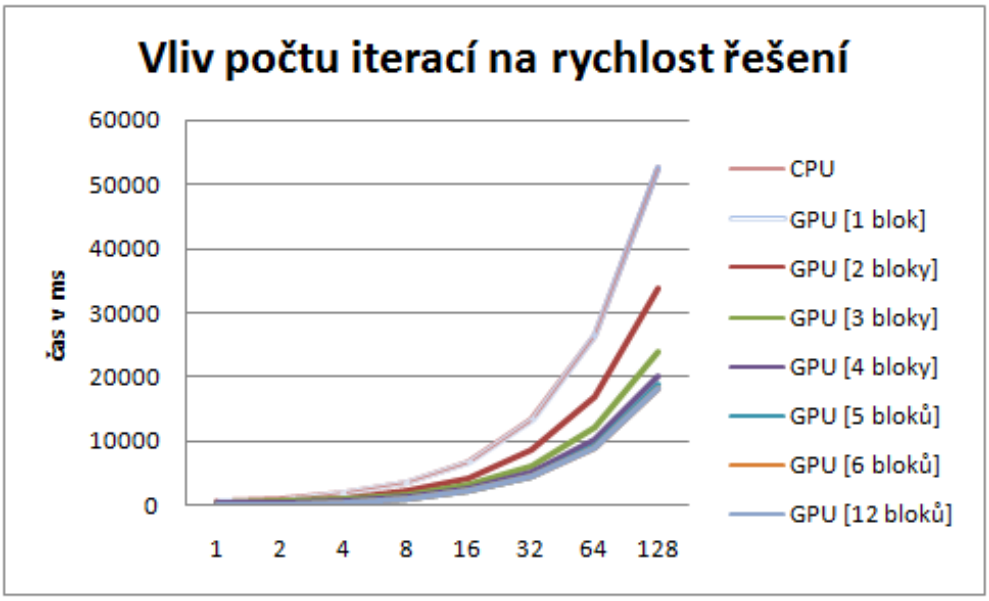
```
1.    while (promenna > 0){
2.          ...
3.          promenna--;
4.    }
```

4 Porovnání výkonnosti sekvenčního a paraleního řešení

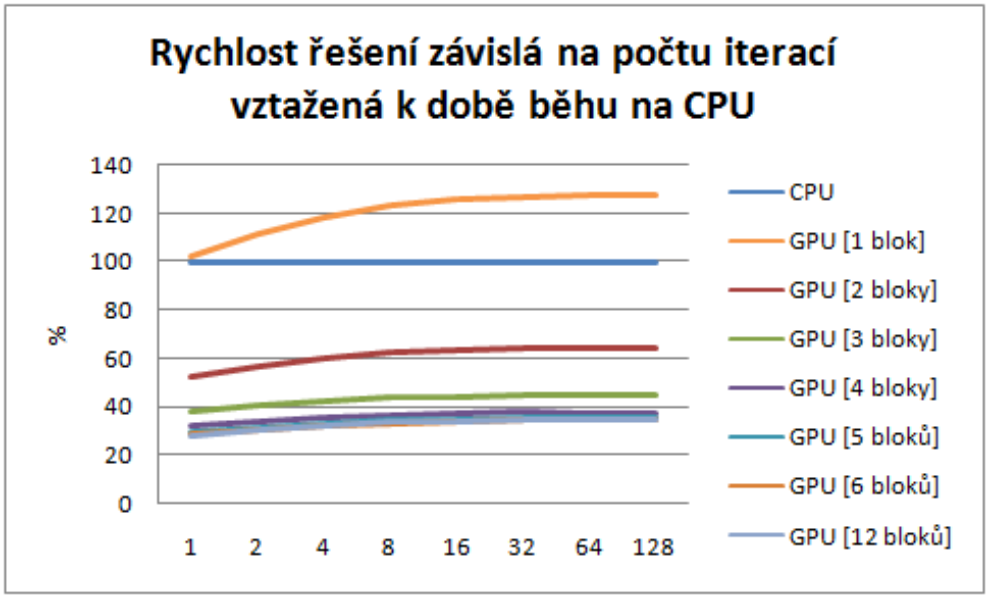
Pro porovnání výkonnosti sekvenční a paralelní implementace jsem použil svůj domácí stroj. CPU bylo dvoujádro bez hyperthreadingu o frekvenci 2.33 GHz. GPU podporovalo CUDA verzi 1.2, počet multiprocesorů 6 o frekvence 1.36 GHz. Obsah dat, na kterých bylo měření prováděno, byl generován náhodně, přičemž jedinou kontrolu nad těmito daty jsem měl ve formě nastavování jejich rozměrů. Pro přesné měření relativního urychlení jsem se rozhodl nalézt takzvaný bod rovnováhy, čímž mám namysli velikosti vstupních dat takové, že rychlost řešení na CPU a GPU je přibližně totožná. Zrychlení bylo vztaženo ke změnám následujících atributů:

- **Počet bloků věnovaných výpočtu**. Myslím si, že je tento atribut důležitý pro zjištění, jak se algoritmus bude chovat na výkonnějších GPU. Počet aktivních bloků, a tím dekonvolvovaných řádků v jeden čas, je totiž závislý na počtu multiprocesorů. Není špatné předpokládat, že vyšší počet multiprocesorů bude jedna z hlavních charakteristik budoucích GPU.
- **Délka řádku výstupního obrazu**. Všechny vnější cykly mají počet iterací roven délce řádku výstupního obrazu.
- **Počet iterací.** Je důležité vědět, jak efektivně je zpracována paralelizace uvnitř iterací upravující výstupní obraz.
- **Délka konvolučního jádra.**

Nalezený bod rovnováhy měl tyto vlastnosti: Délka konvolučního jádra $p = 150$, počet řádků $r = 256$, délka výstupního řádku $n = 2048$, iterací $i = 1$, počet bloků věnovaných výpočtu $b = 1$, počet vláken t v jednom bloku b je 384. Výpočty byly vykonány s variabilním počtem bloků pro zjištění vlivu jejich počtu na celkovou rychlost řešení. V následujících grafech pak zápis *GPU [x bloků]* znamená, že daná křivka popisuje průběh testu spuštěného s *x* bloky vláken.

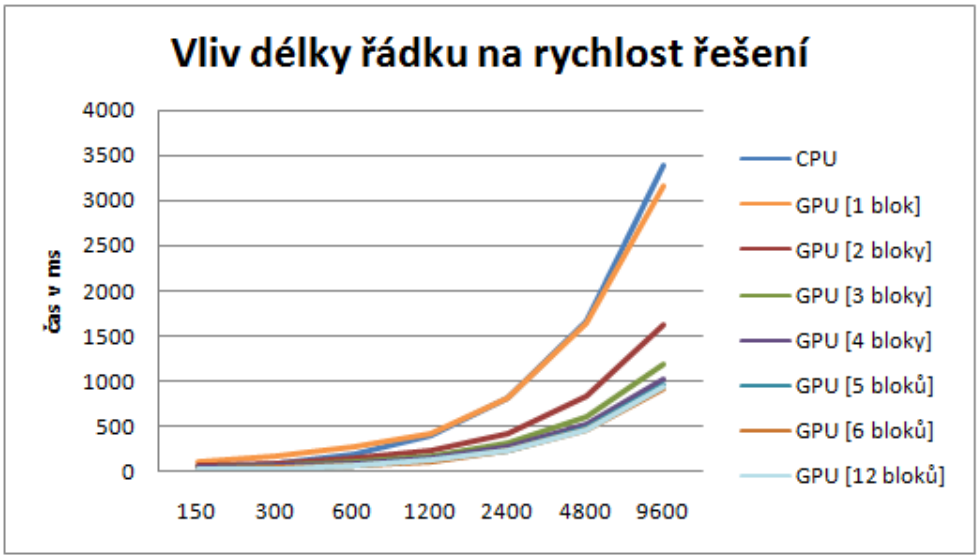


Obrázek 20. Závislost rychlosti výpočtu na počtu iterací.

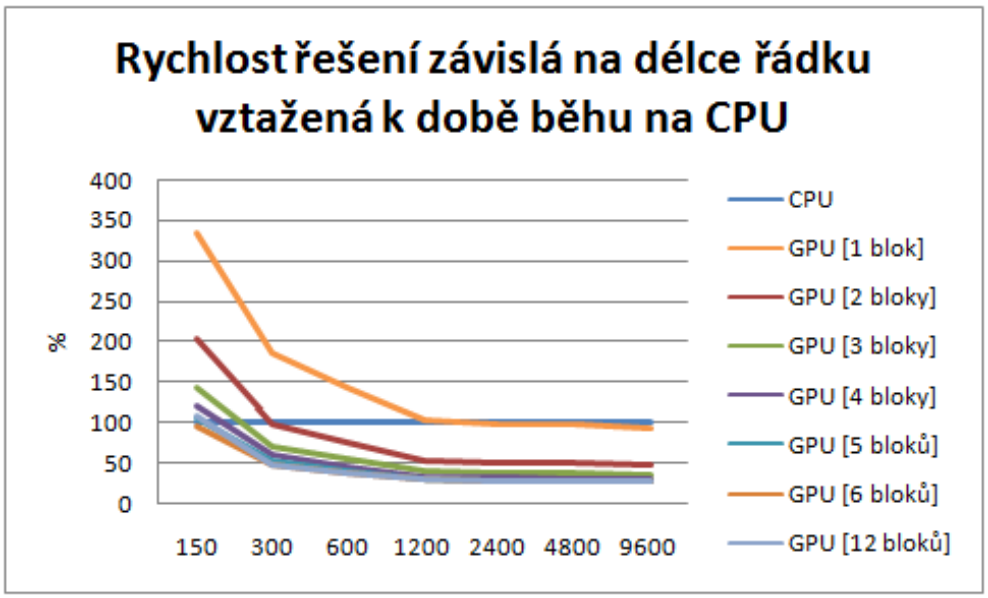


Obrázek 21. Relativní závislost rychlosti výpočtu na počtu iterací vzhledem k době běhu kódu na CPU.

Obrázky 20 a 21 znázorňují zrychlení výpočtu závislé na počtu provedených iterací. Vstupní parametry byly: p=150, r=256, n=2048 a $i \in \langle 1,128 \rangle$. Z rostoucí charakteristiky funkcí popisujících GPU výpočet v obrázku 21 lze usoudit, že paralelní implementace iterací je méně efektivní než implementace sekvenční. Relativní zrychlení je způsoběné obecně vyšším výkonem získaného použitím více bloků vláken, „hrubou silou“. Pro další práci nad tímto algoritmem bych se proto zaměřil na optimalizaci iterací tak, aby byl sklon výše zmíněných funkcí nižší.

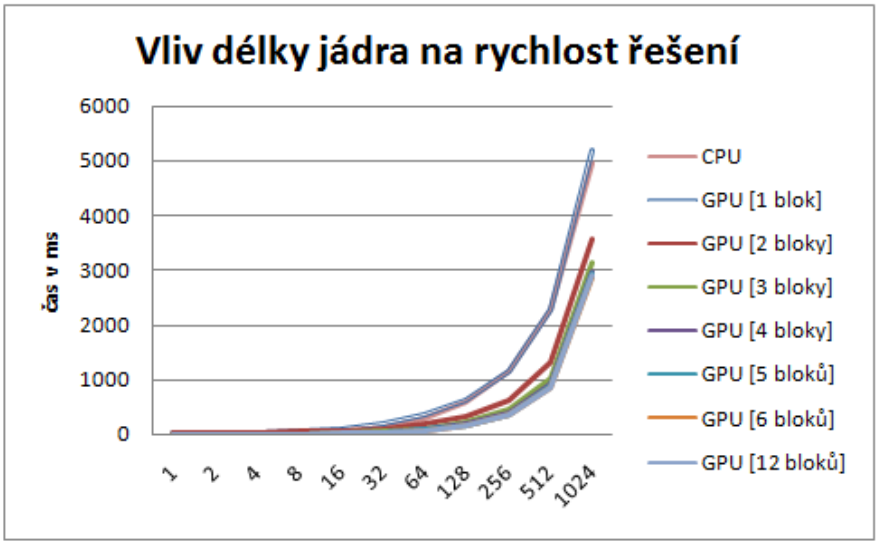


Obrázek 22. Závislost rychlosti výpočtu na šířce obrazu.

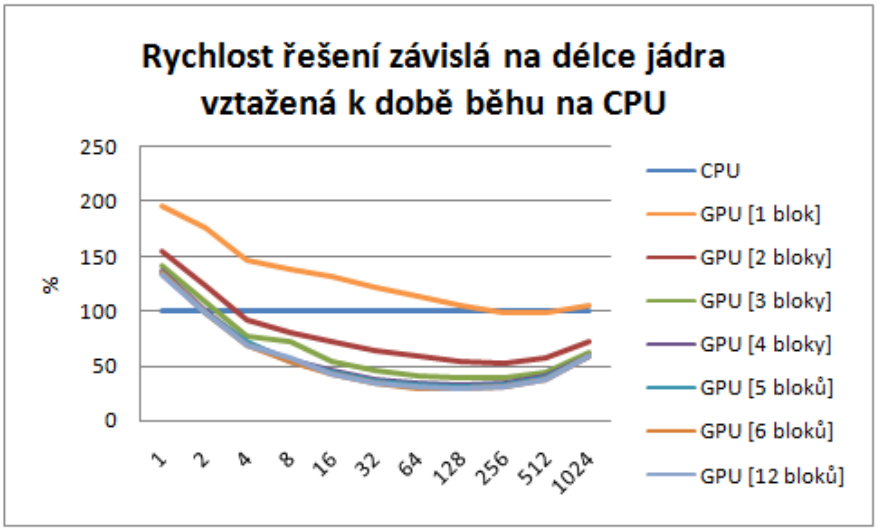


Obrázek 23. Relativní závislost rychlosti výpočtu na délce řádku obrazu vzhledem k době běhu kódu na CPU.

Pro zjištění vlivu velikosti délky řádku na rychlost paralelního řešení byly zvoleny tyto parametry: p=150, r=256, $n \in \langle 150, 9600 \rangle$ a i=1. Z grafu v obrázku 23 lze usoudit, že paralelizace závislá na délce řádku není implementována špatně. Vysoké relativní rozdíly mezi paralelním a sekvenčním řešením při malých hodnotách n je způsobena poměrně vysokou cenou incializace paralelního výpočtu vzhledem k celkové době výpočtu.



Obrázek 24. Závislost rychlosti výpočtu na délce konvolučního jádra.



Obrázek 25. Relativní závislost rychlosti výpočtu na délce konvolučního jádra vzhledem k době běhu kódu na CPU.

Pro zjištění vlivu velikosti délky konvolučního jádra na rychlost paralelního řešení byly zvoleny tyto parametry: $p \in \langle 1{,}1024 \rangle$, $r=256$, $n=2048$ a $i=1$. Vysoké relativní rozdíly mezi paralelním a sekvenčním řešením při malých hodnotách p jsou způsobeny poměrně vysokou cenou incializace paralelního výpočtu vzhledem k celkové době výpočtu. Podle hodnot v grafu v obrázku 25 je vidět, že relativní zrychlení vzhledem k sekvenčnímu výpočtu má pro nižší hodnoty p klesající charakteristiku. Je ovšem také vidět, že pro vyšší hodnoty p je průběh funkce stoupající, což je nutné analyzovat a zjistit, zda-li a jak je možno tento efekt zmírnit.

5 Dekonvoluce v praxi

Pro praktickou ukázku implementace neslepé nezáporné jednorozměrné dekonvoluce jsem jako výchozí obraz zvolil reálnou fotografii zobrazenou v obrázku 15.

Obrázek 15. Původní podklad pro praktickou ukázku aplikace Goldovy dekonvoluce na obraz.

Pro konvoluci výchozího obrázku 15 jsem zvolil konvoluční jádro o 49 prvcích. Obrázek 16 testující aplikaci dekonvoluce je zobrazen níže.

Obrázek 16. Obrázek 15 po aplikaci konvoluce. Tento obrázek sloužil jako objekt, na kterém byl testován algoritmus Goldovy dekonvoluce.

Pro znázornění praktického rozdílu mezi aplikací sekvenčního a paralelního algoritmu běžících po stejnou dobu jsou níže uvedeny obrázky 17 a 18.

Obrázek 17: Obrázek 16 po aplikaci dekonvoluce běžící na CPU po dobu 994 milisekund (17 iterací).

Obrázek 18: Obrázek 16 po aplikaci dekonvoluce běžící na GPU po dobu 992 milisekund (43 iterací).

Rozdíl mezi aplikací sekvenční a paralelní dekonvoluce je patrný. Pro představu o tom, kam dekonvoluce obrázku 16 směřuje zde uvádím v obrázku 19 produkt dekonvoluce obrázku 16 po 1024 iteracích.

Obrázek 19. Ukázka toho, jak nezáporná Goldova dekonvoluce ovlivňuje obraz po aplikaci vysokého počtu iterací (v tomto případě 1024).

6 Závěr

Práce na řešení tohoto problému byla zcela jistě velice zajímavá a poučná, i když přiznávám, že praktické výsledky mohly být lepší, jelikož vzhledem k zadání se mi nepodařilo zpracovat paralelní kód tak, aby bylo možno dekonvoluci řešit v reálném čase. Nicméně doufám, že jsem ve srozumitelné formě představil problém dekonvoluce a nastínil metody jejího řešení s možnostmi dalšího vývoje, a to jak ve smyslu rychlosti, tak i ve smyslu funkcionality.

7 Použitá literatura

[1] Štěpán Šrubař. Enhancing tomograph images by nonnegative Deconvolution. Department of Computer Science, FEI VŠB - Technical University of Ostrava. May 19th 2009.

[2] Štěpán Šrubař. MPRGP in nonnegative deconvolution. Department of Computer Science, FEI VŠB - Technical University of Ostrava. October 2009.

[3] Convolution In *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, [cit. 2010-05-05]. Dostupné z WWW: http://en.wikipedia.org/wiki/Convolution.

[4] Deconvolution In *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, [cit. 2010-05-05]. Dostupné z WWW: http://en.wikipedia.org/wiki/Deconvolution.

[5] Discrete Fourier Transform In *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, , [cit. 2010-05-05]. Dostupné z WWW: http://en.wikipedia.org/wiki/Discrete_Fourier_transform.

[6] Fast Fourier Transform In *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, , [cit. 2010-05-05]. Dostupné z WWW: http://en.wikipedia.org/wiki/Fast_Fourier_transform.

[7] *NVIDIA CUDA Reference Manual* [online]. July 2009 [cit. 2010-05-05]. Dostupné z WWW: http://www.serc.iisc.ernet.in/ComputingFacilities/systems/Tesla_Doc/Nvidia_CUDA_Reference_Manual_2.3.pdf.

[8] *NVIDIA CUDA C Programming Best Practices Guide* [online]. July 2009 [cit. 2010-05-05]. Dostupné z WWW: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf.

[9] *NVIDIA CUDA Programming Guide* [online]. August 26th 2009 [cit. 2010-05-05]. Dostupné z WWW: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

[10] FARBER, Rob. *Dr.Dobb : THE WORLD OF SOFTWARE DEVELOPMENT* [online]. Published on April 15th 2008, Last updated on January 27th 2010 [cit. 2010-05-05]. CUDA, Supercomputing for the Masses. Dostupné z WWW: http://www.drdobbs.com/architecture-and-design/207200659.